1 The Raw LLM API

1.1 An LLM call is a stateless function

At the lowest level, an LLM provider exposes an HTTP endpoint. You POST a JSON document and receive a JSON document. Conceptually:

$$\text{response} = f_{\text{model}}(\text{messages},\ \text{tools},\ \text{parameters})$$

Key idea. The API is **stateless**. The model remembers nothing between calls. Everything it should "know" must be in the request you send this time.

1.2 Anatomy of a request

The now-standard "chat" shape (popularized by OpenAI's API; most providers are similar or offer compatible endpoints):

Listing 1: A minimal chat request (schematic)

```
POST /v1/chat/completions
{
  "model": "some-model-name",
  "messages": [
    {"role": "system",    "content": "You are a concise tutor."},
    {"role": "user",      "content": "What is a token?"}
  ],
  "temperature": 0.2,
  "max_tokens": 200
}
```

Listing 2: A minimal response (schematic)

```
{
  "choices": [{
    "message": {"role": "assistant",
                "content": "A token is a chunk of text ..."},
    "finish_reason": "stop"
  }],
  "usage": {"prompt_tokens": 27, "completion_tokens": 41}
}
```

The pieces worth naming:

- **Roles.** `system` (instructions/persona set by the developer), `user`, `assistant` (previous model outputs), and `tool` (results of function calls; Section 5).
- **Sampling parameters.** `temperature`, `top_p`, `max_tokens`, stop sequences. Lower temperature ⇒ more deterministic output.
- **`finish_reason`.** Why generation stopped: natural end, hit the length cap, or the model requested a tool call. Your code branches on this.

- `usage`. Token counts for billing. Always log these.
- **Streaming.** Optionally the server sends tokens incrementally (server-sent events). Same content, better perceived latency.
- **Structured output.** Many APIs let you supply a JSON schema so the output is guaranteed to parse. Useful for classifiers and extraction.

1.3 Provider differences are mostly cosmetic

OpenAI, Anthropic, Google, and open-model servers (vLLM, Ollama, etc.) differ in field names (e.g. where the system prompt goes, how tool calls are encoded), but share the same model: messages in, message out, optional tools, token-based billing. This is why abstraction layers and frameworks are feasible at all.

2 Tokens and Cost

2.1 What is a token?

Models read and write *tokens*: sub-word chunks produced by a tokenizer (e.g. byte-pair encoding). Rough rule of thumb for English: 1 token $\approx$ 4 characters $\approx$ 0.75 words. Code, non-English text, and numbers often tokenize less efficiently. Billing, context-window limits, and latency are all measured in tokens, not words.

2.2 Input vs. output pricing

Providers charge separately for:

- **Input (prompt) tokens**: everything you send (system prompt, history, tool definitions, retrieved documents).
- **Output (completion) tokens**: what the model generates. Some providers also bill hidden "reasoning" tokens as output.

Output tokens are typically priced several times higher than input tokens (a ratio of roughly 3–5$\times$ is common).

Why the asymmetry? Two phases of inference:

- **Prefill:** the whole prompt is processed in parallel in one big pass. GPUs are efficient at this.
- **Decode:** output is generated *autoregressively*, one token at a time, each step depending on the previous. This is sequential, memory-bandwidth-bound, and occupies the hardware much longer per token.

Hence cheaper to read than to write. A practical consequence: *verbose output is the expensive kind of verbose.*

2.3 The cost formula

For one call, with prices $p_{\text{in}}, p_{\text{out}}$ per million tokens:

$$\text{cost} = \frac{n_{\text{in}}\, p_{\text{in}} + n_{\text{out}}\, p_{\text{out}}}{10^6}$$

Illustrative price table (not real current prices).

Model tier	Input ($/M tok)	Output ($/M tok)
Small / fast	0.25	1.25
Large / capable	3.00	15.00

Example. A request with 1000 input tokens and 300 output tokens:

$$\text{Large: } \tfrac{1000(3)+300(15)}{10^6} = \$0.0075$$
$$\text{Small: } \tfrac{1000(0.25)+300(1.25)}{10^6} = \$0.000625$$

The large model is 12× more expensive here. At 100,000 requests/day that is $750/day vs. about $62/day.

2.4 Why multi-turn conversations get expensive

Because the API is stateless, each turn resends the whole history (Section 3). Let S be the system-prompt tokens, u the tokens per user message, a per assistant reply. The input size at turn t is

$$n_{\text{in}}(t) = S + (t-1)(u+a) + u,$$

so the *cumulative* input over T turns grows **quadratically** in T:

$$\sum_{t=1}^{T} n_{\text{in}}(t) = TS + (u+a)\frac{T(T-1)}{2} + uT.$$

Example. $S = 500$, $u = 100$, $a = 200$, $T = 20$:

$$20(500) + 300(190) + 20(100) = 10{,}000 + 57{,}000 + 2{,}000 = 69{,}000 \text{ input tokens},$$

versus only $20 \times 600 = 12{,}000$ if there were no history. Output is just $20 \times 200 = 4{,}000$ tokens. *Input dominates in long conversations and agent loops*, even though each input token is cheaper.

2.5 Cost levers (checklist)

- **Prompt caching.** Many providers discount (or speed up) repeated *prefixes* of a prompt. Put stable content (system prompt, tool definitions, reference docs) first and variable content last.

- **Cap output.** Set `max_tokens`; instruct the model to be brief; request structured/short answers when appropriate.
- **Manage history** (Section 3).
- **Route** easy work to cheap models (Section 4).
- **Batch APIs.** Asynchronous batch endpoints are often discounted for non-urgent work.
- **Retrieve, don't stuff.** Send only the relevant passages, not the whole corpus.
- **Mind reasoning budgets.** "Thinking" models can spend many hidden output tokens; control the effort setting.

3 Conversation Context and History Management

3.1 The core loop of a chatbot

Since the model is stateless, *your program* owns the conversation state. The simplest possible chatbot:

Listing 3: Minimal chat loop (pseudo-code)

```
history = [ {"role": "system", "content": SYSTEM_PROMPT} ]

while True:
    user_text = input("> ")
    history.append({"role": "user", "content": user_text})

    reply = llm.chat(model=MODEL, messages=history)   # one API call

    history.append({"role": "assistant", "content": reply.text})
    print(reply.text)
```

Key idea. "Memory" is just a list you keep appending to and resending. The **context window** (a hard limit in tokens, covering input *plus* output) bounds how long that list can be.

3.2 What happens when history gets too long?

Three problems arise: (1) you exceed the context window and the call fails, (2) cost grows quadratically (Section 2), (3) quality can degrade as relevant details get buried in a long context. Common strategies:

Sliding window / truncation. Keep the system prompt plus the last k messages. Simple, but forgets early facts.

Summarization. When history exceeds a threshold, ask a (cheap) model to compress old turns into a summary and replace them.

Retrieval / external memory. Store past facts or documents in a database (often a vector store); retrieve only what is relevant to the current question.

Pruning tool outputs. In agents, tool results (web pages, file dumps) are the biggest bloat. Truncate or summarize them before appending.

Scratchpad / notes file. Let the agent write key facts to a persistent store and re-read them, so the live context can be reset.

Listing 4: Summarize-when-long strategy (pseudo-code)

```
def maybe_compact(history, limit_tokens=6000, keep_last=6):
    if count_tokens(history) < limit_tokens:
        return history

    system   = history[0]
    old      = history[1:-keep_last]
    recent   = history[-keep_last:]

    summary = cheap_llm.chat(messages=[
        {"role": "system", "content": "Summarize this conversation. "
                                      "Preserve facts, decisions, open tasks."},
        {"role": "user",   "content": render(old)},
    ]).text

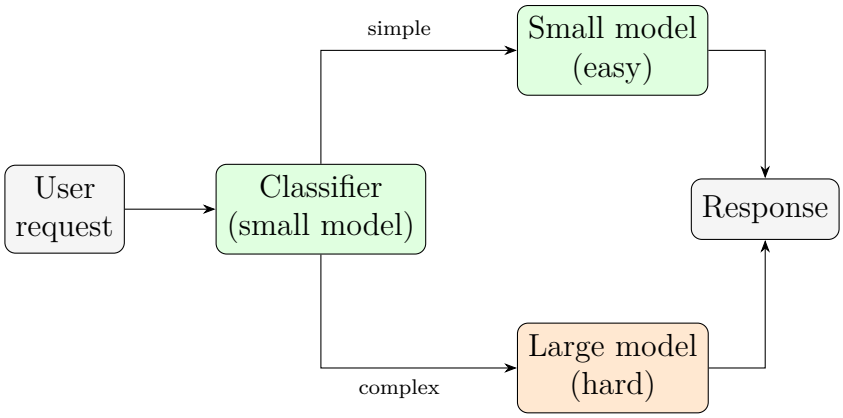
    return [system,
            {"role": "system", "content": "Earlier conversation: " + summary},
            *recent]
```

Watch out. Compaction is lossy and the summarizer may drop a detail that matters later. Also, editing the *beginning* of the history invalidates prompt caches; compact in larger, less frequent steps if caching matters.

4 Cost Optimization by Routing

4.1 Idea

Not all requests need the most capable (and expensive) model. A **router** uses a cheap, fast model to classify the incoming request, then dispatches to an appropriate handler: a small model for easy requests, a large model for hard ones, perhaps a specialized pipeline for others.



4.2 Pseudo-code

Listing 5: A simple LLM router

```
CLASSIFIER_PROMPT = """Classify the request as exactly one of:
SIMPLE  (factual lookup, formatting, short rewrite, small talk)
COMPLEX (multi-step reasoning, math, long code, ambiguous analysis)
Answer with one word."""

def route(request):
    label = small_llm.chat(
        messages=[{"role": "system", "content": CLASSIFIER_PROMPT},
                  {"role": "user",   "content": request}],
        max_tokens=3, temperature=0).text.strip()

    model = LARGE if label == "COMPLEX" else SMALL
    return model.chat(messages=[{"role": "user", "content": request}])
```

4.3 Expected-cost analysis

Let c_s, c_ℓ be per-request cost of the small and large model, c_r the router (classifier) cost, and q the fraction of traffic routed to the small model. Then

$$\mathbb{E}[\text{cost}] = c_r + q\, c_s + (1 - q)\, c_\ell.$$

Example (using the earlier table). Request of 1000 in / 300 out. Classifier sees ≈200 input tokens and emits ≈5 tokens on the small model: $c_r \approx \$0.000056$. With $c_s = \$0.000625$, $c_\ell = \$0.0075$ and $q = 0.8$:

$$0.000056 + 0.8(0.000625) + 0.2(0.0075) \approx \$0.00206,$$

vs. \$0.0075 always-large: a **∼73% saving**. At 100,000 requests/day, about \$206/day instead of \$750/day.

4.4 Variants and pitfalls

- **Cascade (try cheap first).** Run the small model; if a confidence check or validator fails, escalate to the large one. Costs extra when escalation happens, but never misroutes a hard request silently.
- **Rule-based or embedding-based routers.** Keyword rules, or a tiny trained classifier, can replace the LLM classifier and cost almost nothing.
- **Misrouting is the main risk.** A hard problem sent to a weak model yields a confident wrong answer. Measure it: sample routed traffic, grade quality, and track the cost/quality trade-off, not just cost.
- **Router overhead.** It adds latency and a fixed cost. It only pays off if $q\,(c_\ell - c_s) > c_r$ (savings on routed-down traffic exceed router cost).

Key idea. Routing is an application of a general principle: spend expensive compute only where it changes the outcome. Evaluate it empirically.

5 Function (Tool) Calling

5.1 The problem

An LLM only emits text. To check the weather, query a database, or run code, it needs to ask your program to do those things. *Function calling* (a.k.a. tool use) is the protocol for that.

Key idea. The model never executes anything. It emits a structured *request* ("call function `X` with arguments `Y`"). **Your code** executes it and sends the result back as a new message.

5.2 Step 1: describe the tools

You give the model a name, a description, and a JSON Schema for the arguments. The description is effectively a prompt: good descriptions produce good tool selection.

Listing 6: Tool definition (schematic)

```
"tools": [{
  "type": "function",
  "function": {
    "name": "get_weather",
    "description": "Get current weather for a city.",
    "parameters": {
      "type": "object",
      "properties": {
        "city": {"type": "string", "description": "e.g. Boston"}
      },
      "required": ["city"]
    }
  }
```

```
}]
```

5.3 Step 2: the model asks for a call

Given "Should I bring an umbrella in Boston?", the response contains no final answer yet, only a tool request (and `finish_reason` signals this):

```
{"role": "assistant",
 "tool_calls": [{"id": "call_1",
                 "function": {"name": "get_weather",
                              "arguments": "{\"city\": \"Boston\"}"}}]}
```

5.4 Step 3: you execute and reply

```
{"role": "tool", "tool_call_id": "call_1",
 "content": "{\"temp_c\": 11, \"rain_probability\": 0.8}"}
```

You call the API again with the history extended by both messages. The model now answers: "Yes, there is an 80% chance of rain; bring an umbrella."

5.5 Pseudo-code for one tool-using exchange

```
TOOLS = {"get_weather": get_weather}          # name -> real python function

history.append(user_msg)
resp = llm.chat(messages=history, tools=TOOL_SCHEMAS)

if resp.tool_calls:
    history.append(resp.message)              # keep the assistant's request
    for call in resp.tool_calls:
        fn     = TOOLS[call.name]
        result = fn(**json.loads(call.arguments))
        history.append({"role": "tool",
                        "tool_call_id": call.id,
                        "content": json.dumps(result)})
    resp = llm.chat(messages=history, tools=TOOL_SCHEMAS)  # model sees results

print(resp.text)
```

5.6 Practical points

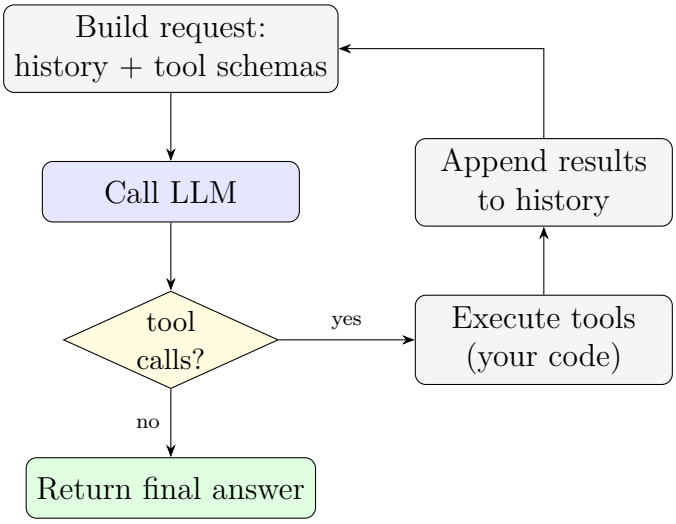
- **Parallel calls.** A model may request several independent calls in one turn; run them concurrently.
- **Validation.** Arguments come from a probabilistic model: validate them against the schema, and return errors as tool results so the model can self-correct.

- **Cost.** Tool definitions are input tokens on *every* call. Fifty verbose tool schemas can dominate your prompt; expose only relevant tools.
- **Safety.** Treat tool arguments as untrusted input. Gate destructive actions (delete, send, pay) behind permissions or human approval. Beware *prompt injection*: text returned by a tool (a web page, an email) may contain instructions aimed at the model.

6 The Agent Loop

6.1 From one tool call to an agent

A single tool call answers one question. An **agent** repeats the pattern: the model decides, acts via tools, observes results, and decides again until it can answer or a limit is hit. The model's output chooses the control flow.



6.2 Pseudo-code

Listing 7: The canonical agent loop

```
def run_agent(task, max_steps=10):
    history = [system_msg(AGENT_INSTRUCTIONS), user_msg(task)]

    for step in range(max_steps):                    # always bound the loop
        history = maybe_compact(history)             # context management
        resp = llm.chat(messages=history, tools=TOOL_SCHEMAS)
        history.append(resp.message)

        if not resp.tool_calls:                      # model chose to finish
            return resp.text

        for call in resp.tool_calls:                 # act
            try:
```

```
            out = TOOLS[call.name](**call.args)
        except Exception as e:
            out = f"ERROR: {e}"                    # let the model recover
        history.append(tool_msg(call.id, truncate(out)))

return "Stopped: step limit reached."
```

6.3 Where everything fits

This one loop is where all earlier topics meet:

- **Context management** happens each iteration, because history grows with every tool result.
- **Cost** compounds: with N steps, input tokens grow roughly quadratically (the same effect as Section 2), so agents can cost 10–100× a single call. Prompt caching helps a lot here, since the prefix is stable across iterations.
- **Routing** can be applied per step (cheap model for simple tool choices, strong model for planning) or per task.
- **Termination and safety:** step limits, token/cost budgets, timeouts, and human approval for risky tools.

6.4 Common patterns built from this loop

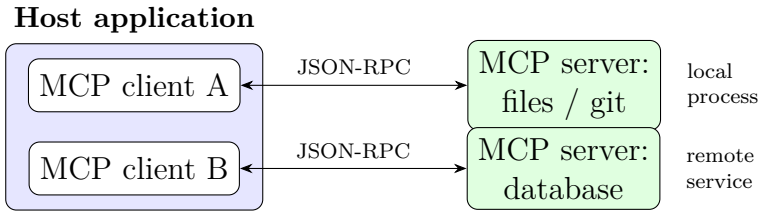
- **ReAct:** interleave reasoning text with tool actions and observations. This is essentially the loop above.
- **Plan-then-execute:** one call produces a plan; subsequent calls execute steps (possibly with cheaper models).
- **Reflection / critic:** a second call reviews the first output and requests revisions.
- **Multi-agent / delegation:** a coordinator agent calls other agents as if they were tools (each with its own context, which also keeps contexts small).
- **Workflows vs. agents:** a *workflow* fixes the sequence of LLM calls in code; an *agent* lets the model decide the sequence. Workflows are more predictable and cheaper; agents are more flexible. Use the simplest thing that works.

7 Model Context Protocol (MCP)

7.1 The integration problem

Without a standard, every application must write custom glue code for every tool or data source: N applications $\times$ M integrations $= N \cdot M$ adapters. MCP is an open protocol (introduced by Anthropic in 2024 and since adopted widely across vendors and frameworks) that standardizes this, so each side implements the protocol once: $N + M$. A common

analogy: "USB-C for AI tools" (or the Language Server Protocol for editors, which inspired it).



7.2 Roles

Host. The user-facing app containing the LLM (a chat app, IDE, or your own agent).

Client. A connector inside the host that maintains a 1:1 connection to one server.

Server. A program exposing capabilities. Can be local (spawned as a subprocess) or remote (over HTTP).

7.3 What a server can expose

- **Tools:** functions the model can call (exactly the function calling of Section 5, but discovered dynamically).
- **Resources:** readable data (files, records, documents) that the host can load into context.
- **Prompts:** reusable prompt templates the user can invoke.

Clients can also offer servers some capabilities in return (e.g. asking the host to run an LLM completion, called *sampling*); details evolve with the spec.

7.4 Transport and message format

Messages are **JSON-RPC 2.0**. Two standard transports: **stdio** (host launches the server as a subprocess and talks over stdin/stdout; typical for local tools) and **HTTP-based streaming** (for remote servers, with authorization). The lifecycle: *initialize* (negotiate versions and capabilities) → *discover* → *use*.

Listing 8: Discovering and calling a tool over MCP (schematic)

```
--> {"jsonrpc":"2.0","id":1,"method":"tools/list"}

<-- {"jsonrpc":"2.0","id":1,"result":{"tools":[
      {"name":"get_weather",
       "description":"Get current weather for a city.",
       "inputSchema":{"type":"object",
                      "properties":{"city":{"type":"string"}},
                      "required":["city"]}}]}}

--> {"jsonrpc":"2.0","id":2,"method":"tools/call",
```

```
     "params":{"name":"get_weather","arguments":{"city":"Boston"}}}

<-- {"jsonrpc":"2.0","id":2,"result":{
      "content":[{"type":"text","text":"11C, 80% chance of rain"}]}}
```

7.5 How MCP plugs into the agent loop

MCP does *not* replace function calling; it standardizes where tool definitions come from and how calls are executed. The host asks each server for `tools/list`, converts the results into the model's native tool schema, and when the model emits a tool call, forwards it as `tools/call`.

Listing 9: MCP inside the agent loop (pseudo-code)

```
servers = [connect("stdio", "python weather_server.py"),
           connect("http",  "https://example.com/db-mcp")]

tool_schemas, owner = [], {}
for s in servers:
    for t in s.list_tools():                # dynamic discovery
        tool_schemas.append(to_llm_schema(t))
        owner[t.name] = s                   # remember which server owns it

# ... inside run_agent(), replace the local TOOLS[...] call with:
result = owner[call.name].call_tool(call.name, call.args)
```

7.6 Trade-offs and security

- **Pro:** reuse of a large ecosystem of servers, decoupling tool authors from agent authors, dynamic discovery.
- **Con:** many connected servers means many tool schemas in the prompt (cost and tool-selection confusion; consider filtering or on-demand loading).
- **Security:** servers are code you run or services you trust. Third-party servers can return malicious content (prompt injection) or have excessive permissions. Apply least privilege, authenticate remote servers, and require confirmation for side-effecting tools.

8 Frameworks and Harnesses

8.1 What is a "harness"?

The *harness* (or framework, or orchestration layer) is the code around the model that implements the loop of Section 6: building requests, running tools, managing context, enforcing limits, logging, retries, and persistence. You can write one in about 50 lines (as we just did). Frameworks exist because production systems also need:

- provider-agnostic model interfaces and retries,

- tool/MCP integration, schema generation from function signatures,
- state, memory, and session persistence,
- multi-agent composition and workflow control,
- tracing, evaluation, guardrails, and deployment tooling.

8.2 Design axes that distinguish frameworks

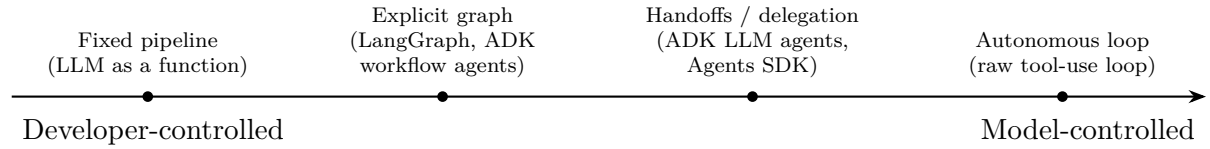
Rather than memorizing products, ask how each answers these questions:

1. **Who controls the flow?** Developer-defined *workflow/graph* (deterministic structure, LLM fills in steps) vs. model-driven *agent loop* (LLM chooses next action).
2. **Abstraction thickness.** Thin (a few primitives; you see the prompts and API calls) vs. thick (many classes, chains, memory types that hide the details). Thick helps prototyping, but hurts debugging and can lag behind provider features.
3. **Explicit vs. implicit state.** Is state a typed object you pass and inspect, or hidden inside agent/memory objects?
4. **Code-first vs. config/declarative.** Python/TypeScript objects and functions, versus YAML/JSON specs or prompt "programs" that are optimized automatically.
5. **Single agent vs. multi-agent composition.** How are agents combined: hierarchy, handoff, or message-passing among peers?
6. **Model/vendor neutrality.** Works equally with any provider, or optimized for one ecosystem and its hosting/deployment?
7. **Production concerns built in.** Observability, evaluation, human-in-the-loop checkpoints, durable execution, deployment.

8.3 Landscape (as of this lecture; verify against current docs)

Framework	Philosophy	Typical strengths / costs
Raw API + your own loop	Maximum transparency and control; no abstraction.	Easiest to debug and reason about; you build persistence, tracing, retries yourself.
LangChain	Composable building blocks (models, prompts, tools, retrievers, output parsers) with a large integration catalog. Modern versions provide a higher-level agent constructor on top.	Fast prototyping, huge ecosystem; layers of abstraction can obscure what is sent to the model.
LangGraph	Agents as an explicit *state graph*: nodes (functions/LLM calls), edges (control flow, conditional), shared typed state, checkpointing.	Fine-grained control, durable execution, human-in-the-loop; more up-front structure to write.
Google Agent Development Kit (ADK)	Code-first toolkit that treats agent development like software engineering. Agents are composable; *workflow agents* (sequential, parallel, loop) give deterministic orchestration, while LLM agents delegate dynamically; first-class sessions/state, evaluation, and a path to Google Cloud deployment. Model-agnostic in principle, tuned for Gemini.	Good structure for hierarchical multi-agent systems and deployment; strongest fit inside the Google ecosystem.
OpenAI Agents SDK	Minimalist: a few primitives (agents with instructions and tools, *handoffs* between agents, *guardrails*, tracing). Model drives the loop.	Small surface area, easy to learn; less opinionated about complex workflows.
Anthropic's Agent SDK / similar "agent harness" SDKs	Expose a battle-tested agent loop (file/shell tools, context compaction, permissions, MCP) as a library; developer supplies instructions and tools.	Strong defaults for long-running, tool-heavy agents; less control over loop internals.
CrewAI, AutoGen/AG2	Multi-agent *role-play*: teams of agents with roles and goals that converse or follow processes.	Intuitive for collaborative decompositions; can be token-hungry and harder to make deterministic.
LlamaIndex	Data-centric: retrieval, indexing, and document pipelines first; agents on top.	Excellent for RAG over private data.

8.4 Three philosophies in one picture



Most real systems are hybrids: a deterministic outer workflow (validation, approvals, retries) around an agentic inner loop where flexibility is valuable. Note that ADK and LangGraph can sit at either end of the spectrum; the spectrum describes *how you use* a framework as much as the framework itself.

8.5 Choosing: a practical heuristic

1. Can a single prompt (with structured output) do the job? Then use a plain API call.
2. Is the sequence of steps known in advance? Use a fixed workflow (code, or a graph/workflow-agent construct).
3. Are the steps unknown and tool-dependent? Use an agent loop, with limits and observability.
4. Do you need multiple specialists, long runs, or auditability? Reach for a framework with state, checkpoints, and tracing.

Watch out. Adopting a heavy framework before understanding the underlying calls is a common source of cost blow-ups and unexplainable behavior. Inspect the actual requests your framework sends (turn on tracing/logging) and read the token counts.

8.6 The same task in three styles (sketch)

Task: "Find the weather and suggest what to wear."

Listing 10: Raw loop: you own everything (see Section 6)

```
answer = run_agent("What should I wear in Boston today?")
```

Listing 11: Graph style (LangGraph-like, schematic)

```
graph = StateGraph(State)                       # explicit typed state
graph.add_node("fetch_weather", fetch_weather) # plain function
graph.add_node("advise",        advise_llm)    # LLM call
graph.add_edge(START, "fetch_weather")
graph.add_edge("fetch_weather", "advise")
graph.add_edge("advise", END)
result = graph.compile().invoke({"city": "Boston"})
```

Listing 12: Agent-object style (ADK / Agents SDK-like, schematic)

```
agent = Agent(
    name="wardrobe_helper",
    model="some-model",
    instruction="Use tools to check weather, then advise on clothing.",
    tools=[get_weather],            # plain python function; schema inferred
)
result = Runner(agent).run("What should I wear in Boston today?")
```

Notice: same primitives (model, tools, loop, state); different answers to "who decides the next step" and "how visible is the state".

9 Summary

- LLM APIs are **stateless** message-in/message-out functions; your code owns the conversation.
- Cost is measured in tokens; **output tokens cost more**, but in long conversations and agent loops **input dominates** because history is resent (quadratic growth).
- Manage context actively: truncate, summarize, retrieve, prune tool output, cache prefixes.
- **Routing/cascades** send easy work to cheap models; always evaluate quality, not just cost.
- **Function calling**: the model requests, your code executes, results return as messages.
- An **agent** is that exchange in a bounded loop; **MCP** standardizes the tool interface; **frameworks** are opinionated harnesses around this loop.
- Distinguish frameworks by **who controls the flow**, **abstraction thickness**, **state explicitness**, and **ecosystem fit**.